A GUIDE TO WRITING DEVICE DRIVERS

JUNE 1984

TABLE OF CONTENTS

FIGURES

ABOUT THIS BOOK

This manual explains how write to programs to control devices for computers running the GRiD Operating System (GRiD-OS).

ASSUMPTIONS ABOUT THE READER

This manual assumes that you are an experienced PASCAL and PL/M programmer and are familiar with the GRiD-OS development environment.

OTHER BOOKS YOU MAY NEED

If you have not programmed under GRiD-OS before, you may need to refer to the following publications:

o GRiD-OS Reference manual for detailed information on operating system calls, semaphores, and file I/O.

o Program Development Guide for information on how to run the compilers and utilities and the GRiDDevelop program.

o Intel PASCAL-86 User's Guide for information on the Pascal programming language.

o Intel PL/M-86 User's Guide for information on the PL/M programming language.

EXAMPLE PROGRAM SOFTWARE

This book contains example programs and develop files. Source
code for these is available on GRiD Central under Software
Subjects 3.1: Contributed Device Drivers.

CHAPTER 1: INTRODUCTION TO DEVICES

WHAT IS A DEVICE?

Here is a diagram of devices in a sample GRiD computer system:
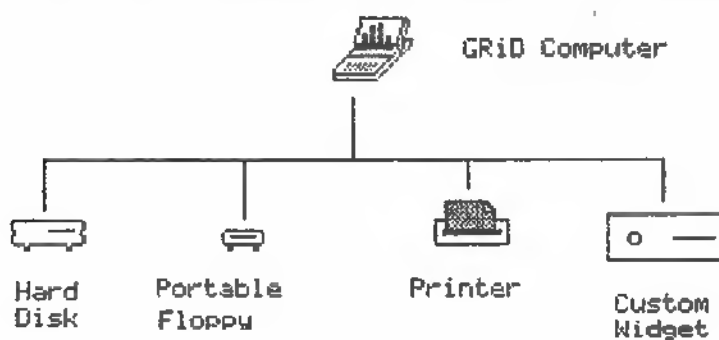
Figure 1-1.   Sample GRiD Computer System

Printers, plotters, digitizers, video disks, laser printers,
laboratory equipment, hard disks, floppy disks, bubble memory and
modems are all physical devices. These devices and others like them
can be controlled by GRiD software.

GRiD-OS supports three types of devices: those that have a serial
interface, those that have a GPIB (IEE-488) interface, and the
internal modem.

HOW DO PROGRAMS COMMUNICATE WITH DEVICES?

Application programs make I/O requests to the GRiD Operating System

(GRiD-OS). GRiD-OS passes the request to a program called a shell which passes the request on to a program called the gateway driver. The gateway driver actually communicates with the device at the hardware level. See Figure 1-2.

```
                    ┌─────────────────┐
                    │   APPLICATION   │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │     GRID-OS     │─────────┐
                    └─────────────────┘         │
                       │         │              ▼
                       │         │        ┌──────────┐
                       │         │        │  SHELL   │
                       │         │        └──────────┘
                       ▼         ▼              │
                                               ▼
              ╭───────╮   ╭───────╮      ╭───────╮
             ╱ SERIAL  ╲ ╱ MODEM   ╲    ╱  GPIB   ╲
            │ GATEWAY   │ GATEWAY    │  │ GATEWAY   │
             ╲ DRIVER  ╱ ╲ DRIVER  ╱    ╲ DRIVER  ╱
              ╰───┬───╯   ╰───┬───╯      ╰───┬───╯
                  ▼           ▼              ▼
            ┌─────────┐ ┌──────────┐  ┌──────────┐
            │ SERIAL  │ │ INTERNAL │  │  GPIB    │
            │ DEVICE  │ │  MODEM   │  │  DEVICE  │
            └─────────┘ └──────────┘  └──────────┘
```
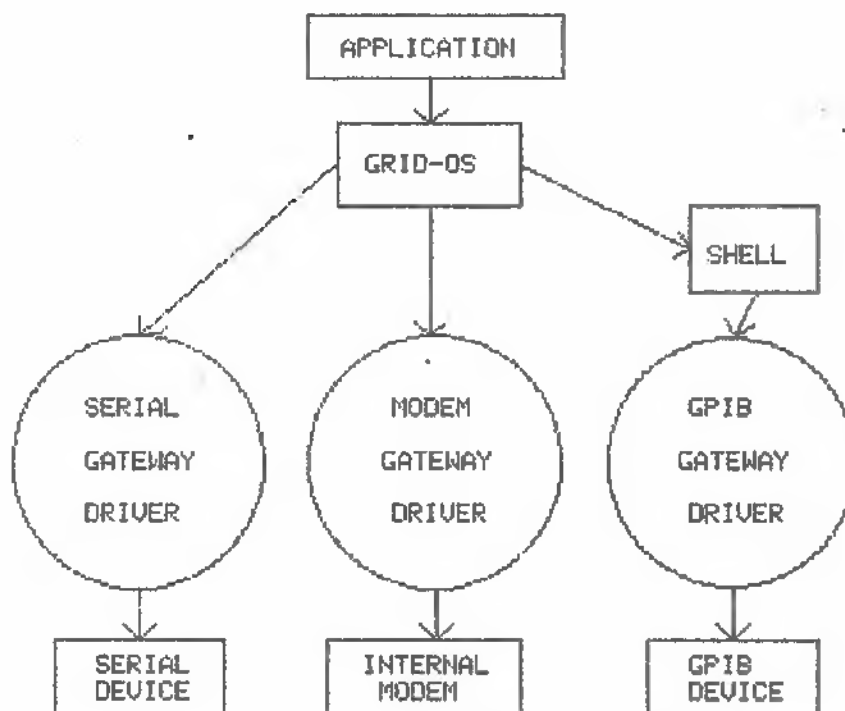
Figure 1-2. Communicating With Devices

GRiD supplies three gateway drivers for device I/O. The serial gateway driver is in a file called Serial~Device~ and the modem gateway driver is in a file called Modem~Device~. The GPIB gateway driver is built into GRiD-OS.

A shell is a program that allows applications to be device independent. You can put all device dependencies in the shell. Then when a new device is used, the application need not be modified.

STEPS FOR WRITING DEVICE PROGRAMS

GRiD supplies all the software necessary to use many devices. To use a device that is not supported by GRiD, or to use the internal modem differently from GRiD applications, you may have to write some software to control the device.

PRINTERS

1. If you don't require text formatting commands or graphics, you can use a generic printer shell. If you do require these features, you should write a shell or modify a generic shell. Writing a shell is covered in Chapter 3 - "Writing A Shell". You may also need to refer to Chapter 5 or Chapter 6 depending on which interface your printer requires. If you write a shell, be sure the final, linked file containing the shell has kind ^Printer^.

2. Use GriDManager's CODE-O command to make the shell your current printer.

THE INTERNAL MODEM

1. Include calls to the modem gateway driver in your application program as described in Chapter 4 - "Programming The Modem Gateway Driver".

2. Activate Modem^Device^ from the command line or from your program. See Chapter 2 - "Adding And Removing Devices" for more information on this.

OTHER SERIAL DEVICES

1. Include calls to the serial gateway driver in your application program for all devices except printers; you must use a shell for printers. See Chapter 5 - "Programming the Serial Gateway Driver"

2. Activate Serial^Device^ from the command line or from your program. See Chapter 2 - "Adding And Removing Devices" for more information on this.

OTHER GPIB DEVICES

1. If it is acceptable to send all device dependent commands from your application, then you can use a generic GPIB shell. If you want your application to be device independent, then you can write or modify a shell. Refer to Chapter 3 - "Writing A Shell" and Chapter 6 - "Programming The GPIB Gateway Driver".

2. Activate your shell from the command line or from your program. See Chapter 2 - "Adding And Removing Devices" for more information on this.

CHAPTER 2: ADDING AND REMOVING DEVICES

This chapter provides information on how to add and remove devices from within your program. You could also use the Activate and Deactivate command line utilities to achieve the same results. These utilities are described in the Program Development Guide.

THE ACTIVE DEVICE TABLE

The Active Device Table, maintained by GRiD-OS, is a table of gateway drivers and device driver shells currently residing in memory and available to be used by an application.

The table associates an ASCII string with the device or shell code; this allows an application to attach to the driver by name instead of by memory address.

You can add or remove (also referred to as activate and deactivate) gateway drivers and driver shells from this table dynamically. This allows you to save memory by keeping memory resident only those drivers necessary at a given time.

You must activate the modem and serial gateway drivers before using them. Because the GPIB gateway driver is in GRiD-OS, it is not necessary to activate it. However, you must activate a GPIB shell before using it.

The active device table also allows you to associate a GPIB address with a shell when it is added to the table. Because driver shells are re-entrant, you can associate the same driver code with devices that have different GPIB addresses. This allows you to avoid duplicating code. For example, the same driver code can be used for a Portable Floppy and an Extra Portable Floppy; only the GPIB address need change.

You can activate a gateway driver or shell from your program by calling a procedure named ActivateDevice. Versions of the procedure exist for both the Compact and Large models of compilation; they can be found in files named Activate Compact~OBJ~ and Activate Large~OBJ~ on GRiD Central: Software Subjects 3.1 under Contributed Device Drivers. The appropriate object file should be linked to the program making the call.

These procedures use the GRiD-OS call OsAddDevice. Applications can use OsAddDevice directly, but the interface to ActivateDevice is simpler and easier to use.

A similar procedure named DeactivateDevice exists in each of the object modules and can be used to remove a driver or shell from the active device table.

THE ActivateDevice PROCEDURE CALL

You must provide information about the attributes of the driver or shell code and its location, which can be one of the following:

o   The driver or shell is in a file on secondary storage.

o   The shell code has been linked to the program making the ActivateDevice call.

o   The shell has already been activated and you want to use the same code for another device.

The ActivateDevice procedure has the following PASCAL declaration.

NOTE: This definition is not in an include file. You must type the declaration yourself in the interface specification of your PASCAL program.

```
PUBLIC  DeviceProcs;
PROCEDURE ActivateDevice   (path: StringPtr;
                            name: StringPtr;
                        VAR entryPoint: BYTES;
                            intAddr: BYTE;
                            attributes: WORD;
                        VAR error: WORD);
```

path   path is a string used to indicate the location of the
                gateway driver or shell to be activated. If you set
                bit 0 of attributes (described below) to 0, set path
                to the pathname of the driver or shell file. If you
                set bit 0 to 1, then set path to the device name

including back quote ('). If the driver or shell code is linked to program (case two), then set path to NIL. If path is not NIL, then the ActivateDevice procedure automatically frees the path StringPtr.

name   name is a string that indicates the name that is to be put into the active device table. Name should NOT have a back quote (') in front of it. If you want the title part of the path parameter to be the name, then set name to NIL. If this parameter is not NIL, then it is automatically freed.

entryPoint   If the shell is linked to the program, then set entryPoint to the shell main procedure name. If not, then this parameter should be a de-referenced NIL Pointer. If entrypoint has any value other than NIL, then path is ignored.

intAddr   intAddr is the interface address (GPIB address) of the device. If the device is not a GPIB device, then set intAddr to NULLBYTE (OFFh).

attributes   Individual bits in this word represent attributes of the device. The following bits are defined (bit 0 is the least significant bit):

Bit	Description
0	Driver location bit. IF 0, then path is the pathname where the driver or shell is located. If 1, then path is the name of an already activated device.
1	Visible bit. 0 – visible; 1 – invisible. Invisible devices don't appear on the file form's device list. For a device to appear on the device list it must be visible and a mass storage device.
2	This bit should be set to zero.
3	Mass storage bit. 0 – the device is a non mass storage device. 1 – the device is a mass storage device.
4	This bit should be set to zero.
5	This bit should be set to zero.
6	This bit should be set to zero.

Searchme bit.
                    0 - the device is searchable
                    1 - the device is not software searchable

                    (For this to be valid the mass storage bit
                    must be 1) Sometimes GRiD-OS searches for files.
                    For example, if you select a text file from a file
                    form, GRiD-OS will search for a file with a kind
                    "Run Text". It will only search in software
                    searchable mass storage devices.

          All other bits in this word are reserved and should be set to
          zero.

     Error   The ActivateDevice routine returns an error code in
                    this parameter.

THE DeactivateDevice PROCEDURE CALL

          To deactivate a gateway driver or shell, call the DeactivateDevice
          routine with the name of the device to deactivate; specify the same
          name an application would use to attach to the driver.

          Here is the PASCAL definition for DeactivateDevice. It should also
          be typed in the interface specification.

          PROCEDURE DeactivateDevice (pathName: StringPtr;
                            VAR error: WORD);

          PathName is automatically freed in the procedure.

CODING EXAMPLES

          Four examples are shown:

          o Activating the internal modem.

          o Activating the Extra Portable Floppy.

          o Deactivating the modem.

          o Activating a linked driver.

ACTIVATING THE INTERNAL MODEM

It is assumed that that a file named Modem~Device~ exists in the
programs subject on a secondary storage device. The ActivateDevice
routine will search each mass storage device until the file is
found.

```
PROCEDURE ActivateTheModem;

  CONST
    nullByte = 0ffh;
    attribute = 80H;

  VAR
    error:   WORD;
    nullPtr: ^BYTE;

  BEGIN
  nullPtr := NIL;
  ActivateDevice (NewStringLit ('Modem~Device~'),
                  NIL,
                  nullPtr^,
                  nullByte,
                  attribute,
                  error);
  END;
```

ACTIVATING AN EXTRA PORTABLE FLOPPY

This example assumes that the Portable Floppy has already been
activated.

```
PROCEDURE ActivateExtraFloppy;

  CONST
    floppyAddr = 7;
    attribute = 9;  ( mass storage, visible, already activated )

  VAR
    error:   WORD;
    nullPtr: ^BYTE;

  BEGIN
  nullPtr := NIL;
  ActivateDevice (NewStringLit ('Portable Floppy'),
                  NewStringLit ('Extra Floppy'),
                  nullPtr^,
                  floppyAddr,
                  attribute,
                  error);
```

```
            END;


DEACTIVATING THE MODEM


        PROCEDURE DeactivateTheModem;

          VAR
            error:   WORD;

          BEGIN
          DeactivateDevice (NewStringLit (''Modem'), error);
                  { Remember the backquote!!! }
          END;


ACTIVATING A LINKED DRIVER


        It is assumed that a generic GPIB driver has been linked to this
        program and that the OsDevice PROCEDURE has been declared PUBLIC in
        the interface specification of this program.


        PROCEDURE ActivateGenericGPIB;

          CONST
             attribute = 80H;
             Address   = 28;

          VAR
            error:   WORD;
            nullPtr: ^BYTE;

          BEGIN
            nullPtr := NIL;
            ActivateDevice (NIL,
                            NewStringLit ('GenericGPIB'),
                            OsDevice,
                            Address,
                            attribute,
                            error);
          END;
```

CHAPTER 3: WRITING A SHELL

A shell is a program that GRiD-OS calls after an application program requests device I/O. Shells are usually written in the PL/M programming language. Shells process the I/O request and then pass the request to a gateway driver which actually communicates with the device.

WHY USE A SHELL?

Shells provide device independence by standardizing the interface to the gateway drivers. Device independence is important because it takes the burden of communicating to different devices away from the applications programmer.

Shells translate generic commands from applications into device-specific commands. For example, GRiDWRite includes commands for formatting printed text. One such command is the boldface command. However, different printers use different control characters for boldface. Rather than requiring GRiDWrite to know the boldface control characters for every printer, GRiDWrite uses a generic printer language which is passed to the shell. The shell then translates the generic boldface command into a printer specific boldface command. In this way GRiDWrite can be used with a new printer by simply creating a new shell rather than changing GRiDWrite.

TYPES OF SHELLS

PRINTERS

For users not interested in text formatting commands or graphics,
generic printer shells are available for serial and GPIB printers.
They pass text as received from applications to the gateway driver.

If text formatting commands or graphics are desired, you can modify
one of the generic shells. GRiD uses a Universal Printer Language
that the shell must interpret and translate into printer specific
commands. See Appendix A for a description of the Universal Printer
Language.

Here are the generic printer shells available on GRiD Central under
Software Subjects 3.0 in the Contributed Programs subject:

MinimumSerial:

>   Protocol: None
>   Baud: 300
>   Assumes printer has at least a 128 byte internal buffer

GenericSerialETX/ACK:

>   Protocol: ETX/ACK
>   Baud: 1200
>   Assumes printer has at least a 128 byte internal buffer

GenericSerialXON/XOFF:

>   Protocol: XON/XOFF
>   Baud: 1200
>   Assumes printer has at least a 128 byte internal buffer

GenericGPIB:

>   Default GPIB address: 21

OTHER GPIB DEVICES

If the device doesn't require data translation, and if
device-specific commands can be done in the application, generic
GPIB shells are available for devices that use GPIB Service Requests
and those that don't. These shells simply pass requests from
applications to the GPIB gateway driver without translating the
information into device specific commands.

If the you need a device specific shell, one of the generic GPIB
shells can be modified.

It is generally not necessary to write or modify a shell for a new serial device other than a serial printer. Instead, the serial gateway driver can be programmed from the application as described in Chapter 5.

WRITING OR MODIFYING A SHELL

An application performs device I/O by making calls to GRiD-OS. The sequence of calls is shown in the following figure:

Attach

Open

Read

Write

GetStatus

SetStatus

Detach

Close
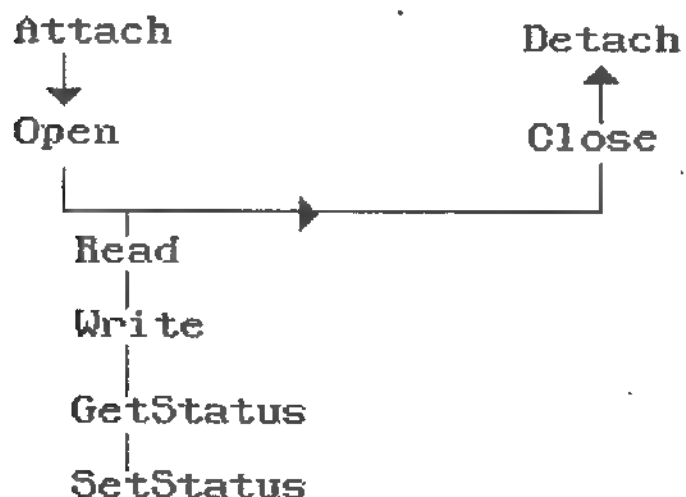
Figure 3-1.   GRiD-OS Device I/O Calls

When GRiD-OS receives an I/O call, it passes an I/O request to the shell. The correspondence between GRiD-OS calls and I/O requests is as follows:

GRiD-OS Call		I/O Request
Activate	—	ddInitialize
OSAttach	—	ddAttach
OSOpen	—	ddOpen
OSRead	—	ddRead
DSWrite	—	ddWrite
OSGetStatus	—	ddGetStatus
OSSetStatus	—	ddSetStatus
OSClose	—	ddClose
OSDetach	—	ddDetach
Deactivate	—	ddDeactivate

Table 3-1.   GRiD-OS calls and I/O Requests

The I/O requests are actually constant numbers that are defined in the PrinterDriver.Inc include file.

SHELL INTERFACE

A shell is written as a PROCEDURE in PL/M that accepts three parameters:

```
OsDevice: PROCEDURE (request, pParameters, pError) PUBLIC REENTRANT;
   DECLARE request     WORD,
           pParamters  POINTER,
           pError      POINTER;
```

Here is a discusion of the parameters passed to the shell:

request   The I/O request number passed to the shell by GRiD-OS.

pParameters A POINTER to a parameter block containing information from the application and GRiD-OS.

pError   A POINTER to a WORD where the shell can return an error code to the application.

Procedure Name

The name of the procedure must be OsDevice. This is because the shell will be linked to an assembly language module (JmpDev.Asm~OBJ~) that expects this name. The assembly language module serves as a main module for the shell; its only purpose is to provide a start address to the linker.

Parameter Block

The parameter block has this format:
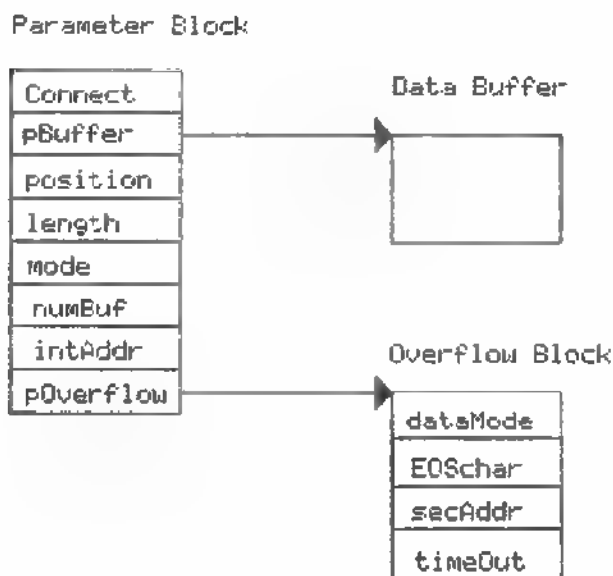
Parameter Block

Figure 3-2. The Parameter Block

Here is the PL/M declaration for the parameter block:

DECLARE ParamListType LITERALLY 'STRUCTURE (

```
        connection       SELECTOR,
        pBufffer         POINTER,
        position         DWORD,
        length           WORD,
        mode             BYTE,
        numBuf           BYTE,
        intAddr          BYTE,
        pOverflow        POINTER)';
```

Do not modify these parameters except as noted below.

pBuffer   A POINTER to the buffer specified by the application when

it made the GRiD-OS call. This buffer contains data for
ddRead and ddWrite requests or status information for
ddGetStatus and ddSetStatus requests.

length   The length of the buffer as set by GRiD-OS according to
               the number of bytes requested by the application; it is
               updated by the gateway driver to reflect the actual number
               of bytes transferred.

intAddr   The GPIB address of the device with which the application
               wants to communicate. If the device is serial, this
               parameter can be ignored.

pOverflow   A POINTER to another STRUCTURE of parameters.

Overflow Block

The format of the overflow block depends on the gateway driver called
and the request being passed to it. For all cases except when a
ddSetStatus is being passed to a gateway driver, the overflow block
will have the following format:

```
DECLARE OverFlowType LITERALLY 'STRUCTURE (

    dataMode        BYTE,
    EOSchar         BYTE,
    secAddr         BYTE,
    timeOut         WORD)';
```

dataMode   The data transfer mode used in GPIB shells. A serial
               shell can ignore this parameter.

EOSchar   The End Of String character used in GPIB shells. A serial
               shell can ignore this parameter.

secAddr   Not used.

timeOut   timeOut is only used by the GPIB gateway driver. It is
               the length of time the gateway driver should wait before
               giving up on a request. This number is in milliseconds
               i.e., a timeout duration of six seconds would be expressed
               as 6000.

When a ddSetStatus request is being passed to a gateway driver, the
format of the overflow block varies for GPIB shells and serial shells.
For information on serial ddSetStatus, see Chapter 5. For information
on GPIB ddSetStatus, see Chapter 6.

A shell communicates with a gateway driver with the OSCallDriver call. The OsCallDriver call looks like this:

```
OsCallDriver (pathName : BYTES;
              level : Byte;
              request : WORD;
            · paramList : ParamListType;
              error : WORD;
```

pathName   The pathname (formatted as a ShortString) of the gateway driver. For a serial device, the pathname is 'Serial. For a GPIB device, the pathname is 'GPIB.

level   A value of 1 specifies that this is a low-level driver (for a mass storage device such as bubble memory, hard disk, or floppy disk), a value of 0 specifies that it is a file level driver (for devices such as printers, PhoneLink, serial devices, or non-disk GPIB devices).

request   A word defining the specific activity (such as open, read, write) that the gateway driver is to perform on the device. This is the dd-request passed by the application to the shell.

paramList   The parameter block specifying device characteristics.

PROCESSING I/O REQUESTS

The body of a typical shell, in outline form, looks like this:

```
IF request = ddInitialize THEN
    Initialize any shell variables.  If necessary, send initialization
    commands to the device using OsCallDriver and a ddWrite request.

IF request = ddRead THEN
    Read from the device through the gateway driver and put data in
    the application's buffer.  The shell should also translate data
    from the device at this point if necessary.

IF request = ddWrite THEN
    Write to the device through the gateway driver from the
    application's buffer.  The shell should should translate data into
    a device-specific format at this point if necessary.

IF request = ddSetStatus THEN
    Set shell characteristics.  A typical use of the OsSetStatus call
    is to allow the application to adjust the device timeout.  This
    ddSetStatus request resulting from OsSetStatus is not passed on to
    the gateway driver.  However, the shell can make separate
    ddSetStatus requests to the gateway driver to set gateway
    characteristics as follows:

    o  Service Request Initialization or Selective Device Clear when
       using the GPIB gateway as described in Chapter 6.

    o  A serial printer shell can set operating characteristics of the
       serial gateway driver like baud rates, stop bits, etc.  In this
       case, the shell should set up an overflow block such as the
       ones used for OsSetStatus discussed in Chapter 5 and pass this
       block and the ddSetStatus request to the serial gateway driver.

IF request = ddGetStatus THEN
    Pass shell status back to the application.  This request is not
    passed on to the gateway driver.

IF request = ddOpen,ddAttach,ddClose,ddDetach,ddDeactivate THEN
    Do nothing.

IF request = any other request THEN
    Return error: RequestNotSupported
```

OsSetStatus

You can use OsSetStatus to pass information to the shell from an
application. The modem and serial gateway drivers are examples of
sophisticated OsSetStatus implementations. In those drivers, you can
pass a number of parameters, including rate of transmission, stop bits,
etc. In the example at the end of this section, OsSetStatus is used
only to set the device timeout. In that example, the OsSetStatus
buffer has the following PL/M declaration:

```
DCL SetStatusType LIT 'STRUCTURE
```

```
        (setStatMode BYTE,
         setNewTime WORD)';
```

OsGetStatus

The shell should meet the GRiD-OS minimum specifications for
OsGetStatus but can add user-defined fields. See the GRiD-OS Reference
Manual for a discussion of OsGetStatus parameters.

EXAMPLE GENERIC SERIAL PRINTER SHELL

This generic serial printer shell has a simple SetStatus function to allow an application to adjust the device timeout. This shell uses the serial defaults for baud rates, stop bits, etc.; the shell could be modified to make ddSetStatus requests to the serial gateway driver.

```
$NOLIST LARGE OPTIMIZE(3)

/* Generic Serial Read and Write Shell
     Default TimeOut: 5 seconds
*/

GenericSerialDriver: DO;

$INCLUDE ('w0'Incs'PlmLit.Inc~Text~)

/* Include declarations for ParamListType, etc.  */
$INCLUDE (PrinterDriver.Inc~Text~)

/* Initialize the parameters in the Overflow Block  */
DCL overflow OverflowType INITIAL (0FFH, 0FFH,
                                        0FFH, 5000H);

OsDevice: PROCEDURE (request, pParams, pError) PUB REENT;
  DCL request WORD;
  DCL pParams PTR;
  DCL pError  PTR;

  DCL error  BASED pError  WORD;
  DCL params BASED pParams ParamListType;

  DCL pSetStatus PTR;
  DCL setStatus BASED pSetStatus SetStatusType;
  DCL StatusBlock SetStatusType;
  DCL getStatus GetStatusType;
  DCL getStatusLength WORD;


  error = 0;

  IF request = ddWrite THEN

    IF params.length > 0 THEN
      DO;
        params.pOverflow = @overflow;
        CALL OSCALLdriver (@(5,''Serial'), 0, DOUBLE (ddWrite),
                            @params, @error);
    END;
  ELSE
  IF request = ddRead THEN
    IF params.length > 0 THEN
      DO;
```

```
                    params.pOverflow = @overflow;
                    CALL OSCALLDriver   (@(5,''Serial'),0,DOUBLE (ddRead),
                                        @params,@error);
                END;
          ELSE
           IF request = ddGetStatus THEN
              DO;
      '* Device is OPEN, update Access allowed, all other 0    */
                 CALL SETB (0, @getStatus, SIZE (getStatus));
                 getStatus.open      = OFFH;
                 getStatus.access    = 4;
                  IF params.length < SIZE (getStatus)
                    THEN getStatusLength = params.length;
                    ELSE getStatusLength = SIZE (getStatus);
                  CALL MOVB (@getStatus, params.pBuffer, getStatusLength);
              END;
          ELSE
           IF request = ddSetStatus THEN
              DO;
                 pSetStatus = params.pBuffer;
                 overflow.timeOut = setStatus.setNewTime;
              END;
          ELSE

      /*  Ignore other valid requests, return error if not valid */

          IF NOT ((request = ddOpen)           OR
                   (request = ddInitialize)    OR
                   (request = ddClose)         OR
                   (request = ddDetach)        OR
                   (request = ddAttach)        OR
                   (request = ddTruncate)      OR
                   (request = ddDeactivate))   THEN
            error = notSupported;
      END;

      END; /* Module */
```

CHAPTER 4: PROGRAMMING THE MODEM GATEWAY DRIVER

If GRiDTerm, GRiDVT100, GRiD3101, GRiDAccess, or GRiDManager do not satisfy your communication requirements, you'll need to write a program that sets the modem options yourself.

GRiD provides a modem gateway driver called Modem~Device~, through which you can control the modem and send and receive data.

The modem gateway driver supports voice mode, where you can talk through the GRiD handset and listen on the speaker, and data mode, where the modem communicates with a remote modem.

COMMUNICATION WITH THE MODEM DRIVER - PROGRAM OVERVIEW

To write a program that reads from and writes to the modem gateway driver:

1. Add Modem~Device~ as an active device.

2. Attach to the modem using OsAttach with pathname = 'Modem.

3. Open the modem using OsOpen.

4. Program desired operating characteristics (bits/sec, stop bits, etc) using OsSetStatus as described in this chapter.

5. If you are originating the call, use OsSetStatus to dial a phone number and to establish a connection. If you are answering, use OsGetStatus to obtain an identification number for the ring semaphore. The gateway driver signals the ring semaphore when it detects an incoming call. You should wait for the signal using OsWait, then use OsSetStatus to connect with the remote modem.

6. Read from and write to the modem using OsRead and OsWrite.

7. Close and detach the modem using OsClose and OsDetach.

8. Remove the modem from the active device table.

THE OsSetStatus CALL

The OsSetStatus call has this PASCAL declaration:

```
PROCEDURE OsSetStatus  (conn:WORD; VAR pBuffer:BYTES;
                        length:WORD; VAR error:WORD);
```

conn   The connection number returned from an OsAttach call.

pBuffer   A pointer to a parameter block. The parameter block has a
            mode byte followed by a varying number of parameter bytes.
            See Figure 4-1.

length   The length of the parameter block including the mode byte.

error   A WORD where an error code is returned. You can examine
            this word to determine if the call was successful.

The parameter block has this format:
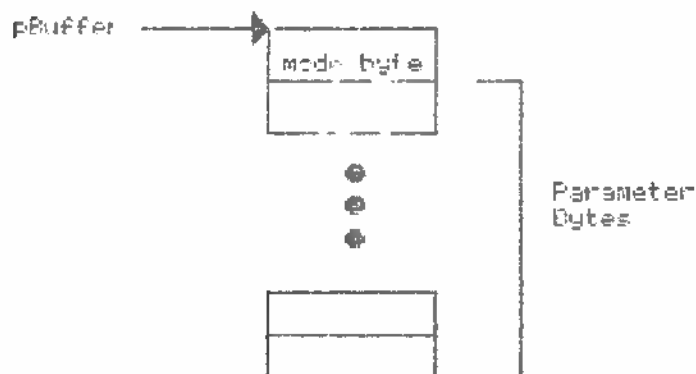
Figure 4-1.   The Parameter Block

OsSetStatus MODES

The following section describes the modes that can be used to program the modem gateway driver.

MODE 0 - RETURN TO DEFAULT SETTINGS

```
---------
:       :
:   0   :
:       :
---------
```

A parameter block with mode byte = 0 and no parameter bytes causes the modem gateway driver to return to a default state:

	Default Value	Mode To Change
Bits/Sec	1200	7
Data Bits	8	1
Stop Bits	2	1
Parity	None	1
Connection Timeout	30 Seconds	2
Character Timeout	Forever	2
Receive Queue	Internal	4
Originate/Answer	Automatic	43
Dial Tone Timeout	15 Seconds	49
Disconnect Timeout	3.5 Seconds	49

MODE 1 - SET OPERATING VALUES

		Bits/	Stop		
1	Protocol	Char.	Bits	Parity	

```
Protocol   = 1        Asynchronus

Bits/Char. = 5    -  5 bits/char.
             6    -  6 bits/char.
             7    -  7 bits/char.
             8    -  8 bits/char.

Stop Bits  = 1    -  1 stop bit
             2    -  1.5 stop bits
             3    -  2 stop bits

Parity     = 0    -  None
             1    -  Even
             2    -  Odd
             3    -  Mark
             4    -  Space
```

MODE 2 - SET TIMEOUT VALUES

2		CharTimeout	ConnectTimeOut

CharTimeOut and ConnectTimeOut are WORD values.

CharTimeOut   The number of milliseconds the gateway driver should
                wait for a character before issuing a TimeOut error.
                If you set CharTimeOut to zero, then the gatway
                driver waits until the requested number of bytes are
                available.

ConnectTimeOut The number of milliseconds the gateway driver should
                wait for a handshake from the other modem after a
                go-to-data (mode six) command.

MODE 3 - FLUSH RECEIVE FIFO BUFFER

```
 ----------
|          |
|    3     |
|          |
 ----------
```

Mode 3 can be used to remove spurious characters from the receive
FIFO buffer.

MODE 4 - SET USER DEFINED RECEIVE FIFO BUFFER

```
 ------------------------------------------------------------
|        |                    |                    |         |
|   4    |      fifoPtr       |     fifoLength     |         |
|        |                    |                    |         |
 ------------------------------------------------------------
```

fifoPtr   A POINTER to an input buffer in the application
                 program.

fifoLength   The length of the new buffer.

Use this mode when you want the gateway driver to use a larger input
buffer than the driver's internal 32 character FIFO buffer to avoid
overflow and loss of data. You should not access the new buffer
directly; but should use the OsRead call instead.

MODE 5 - DISESTABLISH DATA CONNECTION

```
 ----------
|          |
|    5     |
|          |
 ----------
```

Use mode 5 to exit data mode. Voice mode will be entered if it has
been enabled with mode 45; otherwise an idle state will be entered.

MODE 6 - ESTABLISH DATA CONNECTION

```
----------
|        |
|   6    |
|        |
----------
```

Use mode 6 to go to data mode and attempt to handshake with the other modem. If this does not happen within the connection timeout period (specified in mode 2), a TimeOut error will be returned. You can determine the type of handshake with mode 43.

MODE 7 - SET BITS/SECOND

```
-----------------------
|       |             |
|   7   |  Speed      |
|       |             |
-----------------------
```

```
Speed = 5   -   300  Bits/Sec.
        7   -   1200 Bits/Sec.
```

MODE 41 - TAKE PHONE OFF HOOK

```
----------
|        |
|   41   |
|        |
----------
```

Use mode 41 to connect to the phone line. The off-hook function is also done automatically in mode 44.

MODE 42 - PUT PHONE ON HOOK

```
----------
|        |
|   42   |
|        |
----------
```

Use mode 42 to disconnect from the phone line. You cannot reconnect to the phone line for a time called the disconnect delay. This delay ensures that the phone is really hung up. The disconnect

delay can be set with mode 49.

MODE 43 — SET ORIGINATE OR ANSWER

```
-----------------------------------
|             |  Originate |
|     43      |  or        |
|             |  Answer    |
-----------------------------------
```

Originate or Answer = 0   — Automatic
                      1   — Originate
                      2   — Answer
                      255 — Don't Change

Use mode 43 to control the type of handshake used in a mode 6
command. If automatic is chosen, the gateway driver will use
originate mode if the phone has been dialed since the last off hook
command, else answer mode will be used.

MODE 44 - DIAL PHONE NUMBER

```
-----------------------------------------  . . .  --------
|         |  Touch |         |             |           |
|   44    |  Tone  | Length  | Phone Number|           |
|         |        |         |             |           |
-----------------------------------------  . . .  --------
```

Mode 44 causes the modem gateway driver to dial a number. If the
phone is not off the hook (see mode 41), then an off hook command is
automatically executed before dialing.

TouchTone   A BOOLEAN that indicates if a touchtone or pulse
                 dialing is being used. Use TRUE for a touchtone
                 phone.

Length   The length of the phone number that follows.

Phone Number   The phone number to be dialed. The number should be
                 in ASCII form. The following characters are valid:

     Character   Interpretation
     ---------   --------------

       0-9   Same as on a phone
        .   One second delay
        ^   Wait for a dial tone
        *   Same as on a touchtone phone
        #   Same as on a touchtone phone
 space, parentheses, dash   Ignored

Spaces, parentheses and dashes can be used to make the number more
readable but are ignored by the gateway driver. However, they must
be included when determining the length parameter.

MODE 45 - ENABLE VOICE MODE

```
-----------------
|         |
|   45    |
|         |
-----------------
```

Use mode 45 to enable voice mode. When voice mode is enabled, voice
mode will be entered when data mode is exited with mode 5.

MODE 46 - DISABLE VOICE MODE

```
  ------------
  |          |
  |    46    |
  |          |
  ------------
```

If voice mode is disabled, an idle state is entered when exiting
data mode using mode 5.

MODE 49 - SET TIMEOUT VALUES

```
  -----------------------------------------------------------
  |       |                      |                          |
  |  49   |  DialTone Timeout    |  DisconnectTimeOut       |
  |       |                      |                          |
  -----------------------------------------------------------
```

DialTone Timeout and DisconnectTimeout are WORD values.

DialTone Timeout   The number of milliseconds the gateway driver
                        should wait for a dial tone when a caret (^) is
                        encountered in a phone number. If this time is
                        exceeded, a TimeOut error is returned.

Disconnect TimeOut   The number of milliseconds the gateway driver
                        should delay before allowing a reconnection
                        (mode 41).

MODE 50 - SET SPEAKER VOLUME

```
  ---------------------------
  |          |              |
  |    50    |   Volume     |
  |          |              |
  ---------------------------
```

        Volume = 0   - Speaker Off
                  .
                  .

        255 - Maximum Volume

This mode lets you adjust the volume of the speaker.

THE OsGetStatus CALL

The OsGetStatus call obtains information about the current state of
the modem gateway driver. This call is described in the GRiD-OS
Reference Manual but the status record format differs for every
device. The status record for the modem gateway driver has this
format:

```
StatusType = RECORD
                open:              BOOLEAN;
                access:            BYTE;
                seek:              BYTE;
                filePosition:      LONGINT;
                numCharsInFifo:    LONGINT;
                syncDetect:        BOOLEAN;
                connection:        BYTE;
                usartStatus:       BYTE;
                modemStatus:       BYTE;
                RingSID:           WORD;
              END;
```

open   If the modem gateway is attached, this BOOLEAN is
                    TRUE.

access   This BYTE is bit-mapped to indicate the type of
                    access allowed. It will always be set for read and
                    write access. See the GRiD-OS Reference Manual for
                    a description of the bit-map.

seek   This BYTE is always 0.

filePosition   This LONGINT is always 0.

numCharsInFifo   This LONGINT contains the number of characters
                    currently in the recieve buffer.

synchDetect   This BOOLEAN is always FALSE.

connection   This contains the current status of the connection:

                           connection = 0 - No connection established
                                        1 - Off hook, voice mode.
                                        2 - Off hook, data mode.
                                        3 - Not used.
                                        4 - Carrier was lost.

usartStatus   This byte allows you to determine if errors have
               occured on the interface.   The usartStatus byte is
               bit-mapped as follows:

                         MSB   LSB

                          7 6 5 4 3 2 1 0

                          x P F O U x x x

                    where: P marks the bit position for parity error.
                           .F marks the bit position for framing
                    error.
                           O marks the bit position for overrun
                    error.
                           U marks the bit position for underrun
                    error.

               A one (1) in a bit position means that error has
               occured.

modemStatus   This byte allows you to determine the state of
               certain signals on the interface.   The modemStatus
               byte is bit-mapped as follows:

                         MSB   LSB

                          7 6 5 4 3 2 1 0

                          x x x C x D x x

                    where: C marks the bit position for Clear To Send
                    (CTS)
                           D marks the bit position for Dial Tone
                    Detect

               A one (1) in a bit position indicates that signal
               is active.

ringSID   This WORD contains the ring semaphore
               identification number.   The semaphore will be
               signaled when the ring indicator line is active.

PROGRAMMING THE MODEM EXAMPLE

This is an example of a program that originates communication with a
remote modem. It does the necessary setup and then reads a
character from the modem and writes it back. It then does the steps
necessary to clean up.

NOTE: Although this program does not perform error checking after
GRiD-OS calls, you should include error checking in your code to
improve reliablity.

```
$DEBUG COMPACT NOLIST
MODULE  Main;
$INCLUDE ('w0'incs'Common.inc~text~)
$INCLUDE ('w0'incs'ConPas.inc~text~)
$INCLUDE ('w0'incs'OsPasProcs.inc~text~)
$INCLUDE ('w0'incs'OsPasTypes.inc~text~)
$INCLUDE ('w0'incs'WindowProcs.inc~text~)
$INCLUDE ('w0'incs'WindowTypes.inc~text~)
$LIST
PROGRAM Main;

CONST deviceName      = ' 'Modem';
      tempNumber      = '^9'5551212';

TYPE  NumberType = RECORD
          mode:        CHAR;
          touchtone:   BOOLEAN;
          length:      CHAR;
          number:      PACKED ARRAY [1..10] OF CHAR;
      END;

VAR   modemID:         WORD;
      ch:              CHAR;
      ParameterBlock:  PACKED ARRAY [1..9] OF CHAR;
      pathName:        PACKED ARRAY [1..7] OF CHAR;
      reserved:        Byte;
      error:           WORD;
      Phone:           NumberType;
      actual:          INTEGER;

BEGIN

{----------- attach to the modem --------------------------------}

  pathName    := deviceName;
  pathName[1] := CHR(6);     { Device name is 6 characters }
  reserved    := 0;

  modemID := OsAttach (pathName, oldFileMode, reserved,
updateAccess, error);

{----------- open the Modem --------------------------------------}
```

```
    OsOpen (modemID, 1, error);


{----------- now establish some appropriate Modem settings --}

  ParameterBlock[1] := CHR(1);          { mode byte }
  ParameterBlock[2] := CHR(1);          { async }
  ParameterBlock[3] := CHR(8);          { 8 data bits }
  ParameterBlock[4] := CHR(3);          { 2 stop bit }
  ParameterBlock[5] := CHR(1);          { even parity }

  OsSetStatus (modemID, ParameterBlock, 5, error);

{----------- set bits/sec ----------------------------------}

  ParameterBlock[1] := CHR(7);          { mode byte }
  ParameterBlock[2] := CHR(5);          { 300 Bits/Sec }

  OsSetStatus (modemID, ParameterBlock, 2, error);

{----------- turn the speaker up ---------------------------}

  ParameterBlock[1] := CHR(50);     .   { mode byte }
  ParameterBlock[2] := CHR(255);        { volume byte }

  OsSetStatus (modemID, ParameterBlock, 2, error);

{----------- dial the number -------------------------------}

  Phone.mode := CHR(44);        { mode byte }
  Phone.touchtone := TRUE;      { touchtone }
  Phone.length := CHR(10);      { # length  }
  Phone.number := tempNumber;   { number    }

  OsSetStatus (modemID, Phone, 13, error);

{----------- go to data mode -------------------------------}

  ParameterBlock[1] := CHR(6);          { mode byte }

  OsSetStatus (modemID, ParameterBlock, 1, error);

{----------- turn the speaker off --------------------------}

  ParameterBlock[1] := CHR(50);         { mode byte }
  ParameterBlock[2] := CHR(0);          { volume byte }

  OsSetStatus (modemID, ParameterBlock, 2, error);

{----------- read a CHAR from the modem --------------------}

  actual := OsRead (modemID, ch, 1, error);
```

```
{----------- write a CHAR to the modem  --------------------}

  OsWrite (modemID, ch, 1. error);

{----------- disestablish data mode  ----------------------}

  ParameterBlock[1] := CHR(5);          { mode byte }

  OsSetStatus (modemID, ParameterBlock, 1, error);

{----------- put phone on hook ----------------------------}

  ParameterBlock[1] := CHR(42);         { mode byte }

  OsSetStatus (modemID, ParameterBlock. 1, error);

{----------- close the modem ------------------------------}

  OsClose (modemID, error);

{----------- detach from the modem ------------------------}

  OsDetach (modemID,error);

  OsExit (0);

END
  .
```

DEVELOP FILE FOR MODEM EXAMPLE

```
:Name:   Modem Example
:Prefix: Example

:Sources:'
  ModemExample.Pas

:Listings:
   'w'LST'
:Objects:
  'w'OBJ'

:Controls Yes w/Debug: DEBUG

:Link:
  Link 'w'OBJ'ModemExample.Pas~OBJ~,'w'Libs'CompactSystemCalls~Lib~
TO ModemExample~RUN~ BIND SEGSIZE (STACK(+1500)) NOPRINT

:Test:
  Activate 'w'Programs'Modem~Device~
  ModemExample
  Deactivate 'Modem

:Debug:
  Debug ModemExample

:Command Line:
  DevelopmentExecutive

:GridManager:
  GriDManager
```

CHAPTER 5: PROGRAMMING THE SERIAL GATEWAY DRIVER

SERIAL COMMUNICATIONS OVERVIEW

This chapter assumes you are familiar with the RS-232C serial
communications standard. However, a short review of some relevant
concepts follows.

In the RS-232C interface, there are two kinds of communication
equipment - Data Terminal Equipment (DTE) and Data Communication
Equipment (DCE). DTE generally are the source or destination of
communication such as terminals or computers. DCE are usually
devices that provide communication services, such as a modem. See
Figure 5-1.
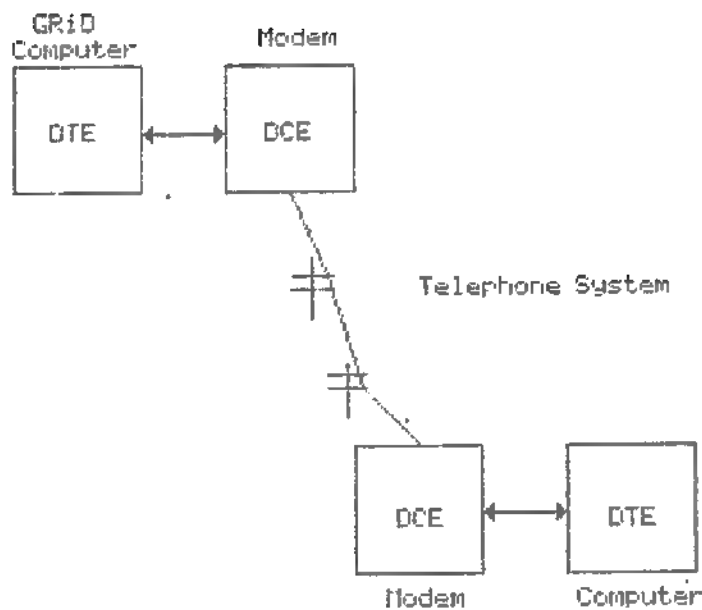
Figure 5-1. The Relationship Between DTE and DCE

The transfer of data between a DTE and a DCE is controlled by certain signals:

Signal	Direction
Request to Send (RTS)	DTE to DCE
Clear to Send (CTS)	DCE to DTE
Data Terminal Ready (DTR)	DTE to DCE
Data Set Ready (DSR)	DCE to DTE
Data Carrier Detect (DCD)	DCE to DTE
Ring Indicator (RI)	DCE to DTE

Table 5-1. Handshaking Signals

THE GRiD SERIAL INTERFACE

The GRiD serial port provides an RS-232C compatible interface. The computer has a serial connector (Canon 2DE19S) on the rear which has 19 pins instead of the standard 25. The six pins not represented are used for a secondary channel but few devices use this feature. GRiD manufactures cables (model #6100) to provide a 19 pin to standard 25 pin connector.

The GRiD computer is wired as a DTE (Data Terminal Equipment).

Therefore, special cables may need to be fabricated in order to connect the computer directly to other devices that are also set up as DTE.

SERIAL CONNECTOR

The serial connector has the following pinout:

PIN	FUNCTION	In/Out	PIN	FUNCTION	In/Out
1	Shield		2	TxD	Out
3	RxD	In	4	RTS	Out
5	CTS	In	6	DSR	In
7	Logic Gnd		8	Carrier Detect	In
9	TxD Reference	Out	10	RxD Reference	In
11	TxC	In	12	DTR	Out
13	Ring Indicator	In	14	-10V DC @ 100mA	Out
15	TxC Reference	Out	16	TxC/Speed Select	Out
17	RxC Reference	In	18	RxC	In
19	+10V DC @ 100mA	Out			

NOTE: The voltage source circuits should be used only for testing.

Table 5-2.   Serial Connector Pinout

RS-423 AND RS-422 COMPATIBILITY

RS-423 and RS-422 are extensions to RS-232 that allow higher signal rates, greater distances between stations, and improved noise immunity. Both standards specify that incoming signals should be evaluated differentially. RS-422 further specifies that signals should be balanced. Balancing is a technique that requires two conductors per circuit but results in better performance.

The GRiD serial interface supports the RS-423 standard and a subset of the RS-422 standard. RS-422 balanced lines are available for TxD, RxD, TxC and RxC. RS-423 compatibility can be attained by grounding the RxC and RxD reference lines.

SERIAL RING SEMAPHORE

Upon initialization, the serial gateway driver creates the serial ring semaphore. The gateway driver signals this semaphore whenever the ring indicator circuit is active (if ring interrupts are enabled). An application program can do an OsWait on the semaphore and allow other processes to run while it is waiting for a ring

indicator signal. You obtain the identification number for the
serial ring semaphore using the OsGetStatus call.

PROGRAMMING THE SERIAL GATEWAY DRIVER FROM AN APPLICATION

1. Add Serial"Device" as an active device.

2. Attach to the serial gateway driver using OsAttach with filename
   = 'Serial.

3. Open the serial gateway driver using OsOpen.

4. Program desired operating characteristics (bits/sec, stop bits,
   etc) using OsSetStatus.

5. Use OsSetStatus to establish a connection with the other device.

6. Read from and write to the serial gateway driver using OsRead and
   OsWrite.

7. Close and detach the serial gateway driver using OsClose and
   OsDetach.

8. Remove the serial gateway driver from the active device table.

PROGRAMMING THE SERIAL GATEWAY DRIVER FROM A SHELL

If you are writing a serial printer shell, you should follow the
steps outlined in Chapter 3 with the following additions:

o In the ddInitialize section of the shell, the shell should add
   serial to the list of active devices as covered in Chapter 2.

o In the ddOpen section of the shell, the shell can make ddRequests
   to the serial gateway driver using OsCallDriver to program
   operating characteristics and to establish a data connection. In
   this case, the overflow block would have the same format as the
   OsSetStatus buffer covered in this chapter.

o In the ddDeactivate section of the shell, the shell should remove
   serial from the list of active devices as covered in Chapter 2.

THE OsSetStatus CALL

The OsSetStatus call has this PASCAL declaration:

```
PROCEDURE OsSetStatus  (conn:WORD; VAR pBuffer:BYTES;
                        length:WORD; VAR error:WORD);
```

conn   The connection number returned from an OsAttach call.

pBuffer   A pointer to a parameter block. The parameter block has a
          mode byte followed by a varying number of parameter bytes.
          See Figure 5-2.

length   The length of the parameter block including the mode byte.

error   A WORD where an error code is returned. You can examine
          this word to determine if the call was successful.

The parameter block has this format:
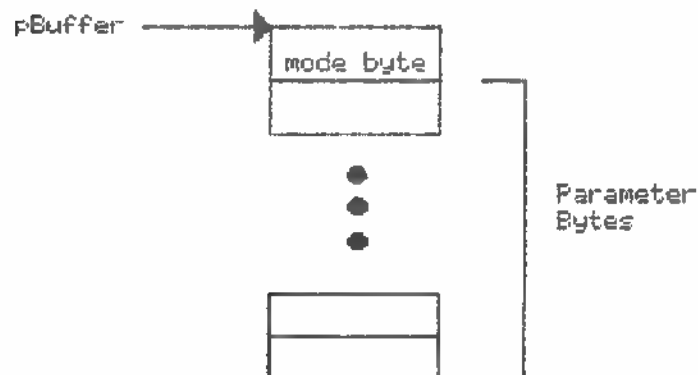
Figure 5-2 The Parameter Block

QsSetStatus MODES

MODE 0 - RETURN TO DEFAULT SETTINGS

```
-----------
|    .    |
|    0    |
|         |
-----------
```

A parameter block with mode byte = 0 and no parameter bytes causes
the gateway driver to return to a default state:

	Default Value	Mode To Change
Bits/Sec	1200	7
Data Bits	8	1
Stop Bits	2	1
Parity	None	1
Connection Timeout	30 Seconds	2
Character Timeout	Forever	2
Receive Queue	Internal	4
Required For Go-To-Data	CTS, DCD	60
Ring Interrupt	Enabled	61

MODE 1 - SET OPERATING VALUES

1	Protocol	Bits/ Char.	Stop Bits	Parity

Protocol   = 1   Asynchronus

Bits/Char. = 5   -   5 bits/char.
             6   -   6 bits/char.
             7   -   7 bits/char.
             8   -   8 bits/char.

Stop Bits = 1   -   1 stop bit
            2   -   1.5 stop bits
            3   -   2 stop bits

Parity   = 0   -   None
            1   -   Even
            2   -   Odd
            3   -   Mark
            4   -   Space

NOTE: Mark and space parity are only allowed with less than eight bits/character.

MODE 2 - SET TIMEOUT VALUES

2	CharTimeout	ConnectTimeOut

CharTimeOut and ConnectTimeOut are WORD values.

CharTimeOut   The number of milliseconds the gateway driver should
                   wait for a character before issuing a TimeOut error.
                   If you set CharTimeOut to zero, then the gatway
                   driver waits until the requested number of bytes are
                   available.

ConnectTimeOut   The number of milliseconds the gateway driver should
                   wait for a handshake from the other device after a
                   go-to-data (mode six) command.

MODE 3 - FLUSH RECEIVE FIFO BUFFER

```
---------
!       !
!   3   !
!       !
---------
```

Mode 3 can be used to remove spurious characters from the receive
FIFO buffer.

MODE 4 - SET USER DEFINED RECEIVE FIFO BUFFER

```
--------------------------------------------------
!       !              !              !           !
!   4   !    fifoPtr   !    fifoLength            !
!       !              !              !           !
--------------------------------------------------
```

fifoPtr   A POINTER to an input buffer in the application
                 program.

fifoLength   The length of the new buffer.

Use this mode when you want the gateway driver to use a larger input
buffer than the driver's internal 32 character FIFO buffer to avoid
overflow and loss of data. You should not access the new buffer
directly; but should use the OsRead call instead.

MODE 5 - DISESTABLISH DATA CONNECTION

```
---------
!       !
!   5   !
!       !
---------
```

Use mode 5 to exit data mode. The driver will drop the DTR and RTS
lines.

MODE 6 - ESTABLISH DATA CONNECTION

```
---------
¦       ¦
¦   6   ¦
¦       ¦
---------
```

Use mode 6 to go to data mode and attempt to handshake with the
other device. The driver will set the RTS and DTR lines active and
wait for DCD and CTS to go active (see mode 60). If this does not
happen within the connection timeout period (specified in mode 2), a
TimeOut error will be returned.

MODE 7 - SET BITS/SECOND

```
-----------------------
¦      ¦         ¦
¦  7   ¦  Speed  ¦
¦      ¦         ¦
-----------------------
```

```
Speed = 0    -    50        Bits/Sec.
        1    -    75
        2    -    110
        3    -    134.5
        4    -    150
        5    -    300
        6    -    600
        7    -    1200
        8    -    1800
        9    -    2000
        10   -    2400
        11   -    3600
        12   -    4800
        13   -    7200
        14   -    9600
        15   -    19200
```

MODE 60 - SIGNALS REQUIRED TO COMPLETE A GO-TO-DATA COMMAND

```
------------------------------------
|        |        |        |        |
|   60   |  Mask  |  Data  |        |
|        |        |        |        |
------------------------------------
```

Mode 60 allows you to control whether CTS or DCD, or both, must be
active to complete a go-to-data command (mode = 6). The default is
that both CTS and DCD must be active. The mask byte is used to
specify which signal(s) are to be affected by this command. The
data byte is used to specify whether the signal(s) are required or
not.

The mask and data bytes are both bit-mapped as follows:

```
    MSB                LSB

    7 6 5 4 3 2 1 0

    x x C x D x x x
```

where "C" marks the bit position for the CTS signal and "D" marks
the bit position for the DCD signal. You should set the bit
position(s) in the mask byte to one for the signal(s) to be changed
and set the bit position(s) in the data byte to one to indicate
which signal(s) need be present to complete a go to data command.

MODE 61 - ENABLE/DISABLE RING INTERRUPT

```
------------------------------
|        |        |          |
|   61   |  Option  |        |
|        |        |          |
------------------------------
```

```
Option byte = 0    - Disable Ring Interrupt
              1    - Enable Ring Interrupt
```

Mode 61 tells the driver whether or not to generate an interrupt
when it detects a ring indicator signal. Some devices may not have
the capability to drive the ring indicator line and will simply keep
it active, thus reducing performance by causing unwanted interrupts.
You should disable the ring interrupt in that case.

THE OsGetStatus CALL

The OsGetStatus call obtains information about the current state of
the serial gateway driver. This call is described in the GRiD-OS
Reference Manual but the status record format differs for every
device. The status record for the serial gateway driver has this
format: '

```
StatusType = RECORD
                open:           BOOLEAN;
                access:         BYTE;
                seek:           BYTE;
                filePosition:   LONGINT;
                numCharsInFifo: LONGINT;
                syncDetect:     BOOLEAN;
                connection:     BYTE;
                ModemControl:   BYTE;
                unused:         BYTE;
                RingSID:        WORD;
            END;
```

open   If the serial gateway is attached, this BOOLEAN
                will be TRUE.

access   This BYTE is bit-mapped to indicate the type of
                access allowed. It will always be set for read and
                write access. See the GRiD-OS Reference manual for
                a description of the bit-map.

seek   This BYTE is always 0.

filePosition   This LONGINT is always 0.

numCharsInFifo This LONGINT contains the number of characters
                currently in the recieve buffer.

synchDetect   This BOOLEAN is always FALSE.

connection   This contains the current status of the connection:

                connection = 4 - Carrier was lost.
                             2 - Connection established.
                             0 - No connection established.

modemControl   This byte allows you to determine the if certain
                      signals are active on the interface.   The
                      modemControl byte is bit-mapped as follows:

                            MSB   LSB

                             7 6 5 4 3 2 1 0

                             x x x x x D C R

                      where: D marks the bit position for DCD
                             C marks the bit position for CTS
                             R marks the bit position for DSR

                      A one (1) in a bit position indicates that signal
                      is active.

ringSID   This word contains the ring semaphore
                      identification number.   This semaphore will be
                      signaled when the ring indicator line is active and
                      ring interrupts are enabled.   See OsSetStatus mode
                      61.

PROGRAMMING THE SERIAL GATEWAY DRIVER EXAMPLE

This is an example of a program that communicates with another device using the serial port. It does the necessary setup and then reads a character and writes it back. It then does the steps necessary to clean up.

NOTE: Although this program does not perform error checking after GRiD-OS calls, you should include error checking routines in your code to improve reliability.

```
$DEBUG COMPACT NOLIST
MODULE  Main;
$INCLUDE ('w0'incs'Common.inc~text~)
$INCLUDE ('w0'incs'ConPas.inc~text~)
$INCLUDE ('w0'incs'OsPasProcs.inc~text~)
$INCLUDE ('w0'incs'OsPasTypes.inc~text~)
$LIST
PROGRAM Main;

CONST deviceName       = ' 'Serial';

VAR   SerialID:        WORD;
      ch:              CHAR;
      ParameterBlock:  PACKED ARRAY [1..9] OF CHAR;
      pathName:        PACKED ARRAY [1..8] OF CHAR;
      reserved:        Byte;
      error:           WORD;
      actual:          INTEGER;

BEGIN

{------------ attach to the serial gateway ----------------------}

  pathName    := deviceName;
  pathName[1] := CHR(7);     { Device name is 7 characters }
  reserved    := 0;

  SerialID := OsAttach (pathName, oldFileMode, reserved,
updateAccess, error);

{------------ open the serial gateway ---------------------------}

  OsOpen (serialID, 1, error);

{---- now establish some appropriate serial driver settings -}

  ParameterBlock[1] := CHR(1);            { mode byte }
  ParameterBlock[2] := CHR(1);            { async }
  ParameterBlock[3] := CHR(8);            { 8 data bits }
  ParameterBlock[4] := CHR(3);            { 2 stop bit }
```

```
      ParameterBlock[5] := CHR(1);              { even parity }

      OsSetStatus (SerialID, ParameterBlock, 5, error);

   {--------------- Set bits/sec ----------------------------------}

      ParameterBlock[1] := CHR(7);        { mode byte }
      ParameterBlock[2] := CHR(5);        { 300 Bits/Sec }

      OsSetStatus (SerialID, ParameterBlock, 2, error);

   {----------- go to data mode -----------------------------------}

      ParameterBlock[1] := CHR(6);        { mode byte }

      OsSetStatus (SerialID, ParameterBlock, 1, error);

   {----------- read a CHAR from the Serial gateway ------------}

      actual := OsRead (SerialID, ch, 1, error);

   {----------- write a CHAR to the serial gateway -------------}

      OsWrite (SerialID, ch, 1, error);

   {----------- disestablish data mode  -----------------------}

      ParameterBlock[1] := CHR(5);        { mode byte }

      OsSetStatus (SerialID, ParameterBlock, 1, error);

   {----------- close the serial gateway  --------------------}

      OsClose (serialID, error);

   {----------- detach from the Serial gateway ----------------}

      OsDetach (SerialID,error);

      OsExit (0);

   END
```

DEVELOP FILE FOR SERIAL EXAMPLE

```
:Name:  Serial Example
:Prefix: Example

:Sources:
  SerialExample.Pas

:Listings:
   'w'LST'
:Objects:
  'w'OBJ'

:Controls Yes w/Debug: DEBUG

:Link:
  Link 'w'OBJ'SerialExample.Pas'OBJ', 'w'Libs'CompactSystemCalls'Lib'
TO SerialExample'RUN' BIND SEGSIZE (STACK(+15,0)) NOPRINT

:Test:
  Activate 'w'Programs'Serial'Device'
  SerialExample
  Deactivate 'Serial

:Debug:
  Debug SerialExample

:Command Line:
  DevelopmentExecutive

:GridManager:
  GriDManager
```

CHAPTER 6: PROGRAMMING THE GPIB GATEWAY DRIVER

GPIB OVERVIEW

The General Purpose Interface Bus (GPIB) is a parallel interface used to transmit byte-wide data. At GRiD, the interface is used for disk drives, printers, and plotters. Since the bus is arbitrated, many devices can be hooked up simultaneously and each device has a unique address. The GPIB supports addresses ranging from 0 to 31.

The interface supports three kinds of devices: Controllers, Talkers and Listeners. A Controller is a device that arbitrates the bus; talkers write data onto the bus; and Listeners only receive data. Some devices may combine the functions. A computer running GRiD-OS is a Controller but also has Talker/Listener capability.

Data transfer on the bus is arbitrated by the Controller. The Controller first places the addresses of a Talker and Listener(s) on the bus before each message. The messages can either be transmitted a byte at a time or in blocks. Handshaking on the bus ensures that Talkers send at a rate compatible with Listeners and there is a signal called EOI (End or Identify) that indicates when a message is complete.

Some devices can interrupt the Controller. This occurs on a line called Service Request (SRQ). After the interrupt, the Controller initiates a process called a Serial Poll to determine which device requested service.

GRiD GPIB GATEWAY DRIVER OVERVIEW

The GPIB gateway driver transmits data in two modes: low speed and
high speed. In the low speed mode (sometimes called the interrupt
mode), the gateway driver transmits data a single byte at a time and
checks after every byte to see if the transmission has been
terminated. Termination can occur one of five ways:

o The number of bytes requested has been transferred.

o The sending device sent a special byte called the end of string
   (EOS) character. For example, a digitizer might send a carriage
   return after each coordinate has been transmitted over the bus.
   The EOS character is device dependent and can be specified to the
   GPIB gateway driver. If the device doesn't have an EOS character
   or if you want to disable this feature, the EOS character should
   be specified as OFFH when programming the gateway driver.

o The other device indicated it was finished by asserting the EOI
   line on the bus.

o The request timed out.

o An error occured on the GPIB bus.

In the high speed mode (sometimes called the DMA mode), the gateway
driver transfers blocks of data to or from a special address in
memory; special timing is implemented to speed up the transfer.
Termination can occur one of four ways:

o The number of bytes requested has been transmitted.

o The other device indicated it was finished by asserting the EOI
   line on the bus.

o The request timed out.

o An error occured on the GPIB bus.

Because there is no EOS concept in the high speed mode, you should
use the low speed mode whenever an EOS function is desired and the
high speed mode otherwise.

If you use the low speed mode, you can program the amount of time
the driver should wait before issuing a request timed-out error. In
the high speed mode, the timeout duration is fixed at five seconds
because a hardware timer is used.

DATA STRUCTURES

Three main data structures are important to a programmer interested in writing a GPIB shell: a parameter block and two kinds of overflow blocks.

PARAMETER BLOCK

The parameter block has this PL/M declaration:

```
DCL ParamListType LITERALLY 'STRUCTURE (

            connection      SELECTOR,
            pBuffer         POINTER,
            position        DWORD,
            length          WORD,
            mode            WORD,
            numBuf          BYTE,
            intAddr         BYTE,
            pOverflow       POINTER)';
```

These parameters shouldn't be modified by the shell unless otherwise noted. The parameters of interest are:

pBuffer   A POINTER to the buffer specified by the application
              when it made the GRiD-OS call.

length   The number of bytes requested by the application. It
              is updated by the gateway driver to the actual number
              of bytes transferred.

intAddr   The GPIB address of the device with which the
              application wants to communicate. If the shell was
              not assigned an address when it was activated, this
              parameter is NULL (OFFH). You should check for NULL
              and assign an address in that case.

pOverflow   A POINTER to another block of parameters. You should
              set pOverflow to point to this block. The format of
              the overflow block varies as described in the
              following sections.

OVERFLOW BLOCK FOR I/O REQUESTS

This overflow block is used when sending ddRead, ddWrite, or
ddDeactivate to the gateway driver; it appears as follows:

```
DCL OverflowType LITERALLY ' STRUCTURE (
```

```
                             dataMode        BYTE,
                             EOSchar         BYTE,
                             secAddr         BYTE,
                             timeOut         WORD)';
```

dataMode   The data transfer mode. You should put a 0 in this
                byte if the high speed mode is desired, and a 2 if
                the low speed mode is desired.

EOSchar   In the low speed mode, set this parameter to the
                character, if any, used to terminate a message. In
                the high speed mode, or if no EOS character is
                desired, set it to OFFH.

secAddr   Not used.

timeOut   In the low speed mode, set this parameter to the
                length of time you want the gateway to wait before
                giving up on a request. This number is in
                milliseconds, i.e., a time out duration of six
                seconds is specified as 6000. If the high speed mode
                is chosen, the timeout is fixed at five seconds.

OVERFLOW BLOCK FOR SETSTATUS REQUESTS

        This overflow block is used when sending a ddSetStatus request. It
        appears as follows:

            DECLARE gateWaySetStatus STRUCTURE (
                        mode   BYTE,
                        dataWord   WORD);

        mode   Set this byte to 0 if you want to send a selective
                        device clear (reset) to the device. You should set
                        this byte to 2 when notifying the gateway driver to
                        recognize service Requests from a device.

        dataWord   If you want to recognize service requests from a
                        device, set this word to the identification number of
                        the semaphore that is to be signaled. When doing a
                        selective device clear, ignore this parameter.

        NOTE: ParamListType and OverflowType are defined in an Include file
              -- PrinterDriver.Inc. GateWaySetStatus should be defined in
              the shell.

SERVICE REQUESTS

Some devices can interrupt the computer by asserting the Service Request line. Typically, service requests are used by a device to indicate a readiness to transfer data or to report an error. If the gateway driver has been notified to recognize service requests for a device and one occurs, the driver will:

1. Poll the bus to determine which device requested service.

2. Read a status byte from the requesting device.

3. Signal a semaphore and pass the status byte through the semaphore note. You must create the semaphore and pass the semaphore identification number to the gateway driver with ddSetStatus request, mode two.

There are two major uses for service requests:

o The service request is used to report a readiness to transfer data. Some devices require a command to be written to them before they can transfer data. For example, a device might need a command sent to it before it will transmit. In the ddRead section of the shell, you would write the command to the device and then wait for a service request before reading from the device.

o The service request is used to report an asychronous event. In this case, the application should check for the signal. For example, the device might generate a service request when it detects an error condition. Your application could have a process waiting for an error signal and then take appropriate action.

Service requests are very device-dependent and many devices do not support them. If they do, they may need to be programmed to supply a service request. Check your device manual for details.

When the shell receives a ddDeactivate request, delete the semaphore and pass a ddDeactivate request to the gateway driver. This will inform it to stop responding to service requests from this device.

PROGRAMMING THE GPIB GATEWAY DRIVER EXAMPLES

This section contains four examples of GPIB driver programming:

o Sending a selective device clear to a device.
o Notifying the GPIB gateway driver to recognize service requests from a
   device.
o Reading from the device.
o Writing to the device.

All the examples assume these data declarations:

```
DECLARE params    ParamListType;
DECLARE overflow  OverflowType;
DECLARE plotBuf (8) BYTE;
DECLARE gateWaySetStatus STRUCTURE (
            mode           BYTE,
            dataWord       WORD);
```

RESET THE DEVICE

This code tells the GPIB gateway driver to send a selective device clear
(reset) to the device at the address specified.

```
/* If the device wasn't assigned an address when
   it was activated, assign a default one now */
IF params.intAddr = OFFH THEN
    params.intAddr = 28;

gateWaySetStatus.mode = 0; /* Reset */
params.pOverflow = @gateWaySetStatus;
CALL OSCALLDriver (@(5,''GPIB'),0,DOUBLE (ddSetStatus),
                   @params,@error);
```

This example is for an HP 7470A plotter that requires an Input Mask (IM) instruction before it will assert a SRQ. IM is an HP 7470A plotter specific command. Other devices may require different commands.

```
/* If the device wasn't assigned an address when
   it was activated, assign a default one now */
IF params.intAddr = OFFH THEN
    params.intAddr = 28;

gateWaySetStatus.mode = 2;  /* Set SRQ mode */

/*  Create a semaphore to pass to the gateway driver */
gateWaySetStatus.dataWord = OsCreateSemaphore (@error);
IF error = 0 THEN
DO;
   params.pOverflow = @gateWaySetStatus;
   CALL OSCALLDriver (@(5,''GPIB'),0,DOUBLE (ddSetStatus),
                         @params,@error);

   /* Send the 7470 a command to enable SRQ */
   plotBuf = 'IM,223,4';
   params.pBuffer = @plotBuf;
   params.length = 8;
   params.pOverflow = @overflow;
   CALL OSCALLDriver (@(5,''GPIB'),0,DOUBLE (ddWrite),
                         @params,@error);
END;
```

READ FROM THE DEVICE

This example assumes the device has GPIB address 28, uses the low speed transfer method and terminates messages with a carriage return (0DH).

```
Overflow.dataMode = 2;   /* Low Speed mode */
Overflow.EOSchar  = 0DH; /* Terminate message on CR */
params.pOverflow = @Overflow;
/* If the device wasn't assigned an address when
   it was activated, assign a default one now */
IF params.intAddr = OFFH THEN
    params.intAddr = 28;
CALL OSCALLDriver (@(5,''GPIB'),0,DOUBLE (ddRead),
                      @params,@error);
```

WRITE TO THE DEVICE

This example assumes the device has GPIB address 28, and uses the high speed transfer method.

```
Overflow.dataMode = 0;      /* High Speed mode */
Overflow.EOSchar  = OFFH; /* No EOS char in this mode */
params.pOverflow = @Overflow;
/* If the device wasn't assigned an address when
   it was activated, assign a default one now */
IF params.intAddr = OFFH THEN
    params.intAddr = 28;
CALL OSCALLDriver (@(5,''GPIB'),0,DOUBLE (ddWrite),
                      @params,@error);
```

GENERIC GPIB SHELL EXAMPLE

```
$NOLIST LARGE OPTIMIZE(3)

/* Generic GPIB Read and Write Shell
     Default GPIB address: 28.
     No Service Requests.
     High Speed data transfer mode.
     Default TimeOut: 5 seconds -- Fixed in High Speed Mode.
*/

GenericGPIBDriver: DO;
$INCLUDE ('w0'Incs'PlmLit.Inc~Text~)
$INCLUDE (PrinterDriver.Inc~Text~)

DCL defaultAddress LIT '28';
DCL HiSpeedMode    LIT '0';

/* Initialize the parameters in the Overflow Block  */
DCL overflow OverflowType INITIAL (HiSpeedMode, OFFH,
                                   OFFH, OFFFFH);

OsDevice: PROCEDURE (request, pParams, pError) PUB REENT;
  DCL request WORD;
  DCL pParams PTR;
  DCL pError  PTR;

  DCL error  BASED pError  WORD;
  DCL params BASED pParams ParamListType;

  DCL pSetStatus PTR;
  DCL setStatus BASED pSetStatus SetStatusType;
  DCL StatusBlock SetStatusType;
  DCL getStatus GetStatusType;
  DCL getStatusLength WORD;

  IF request = ddWrite THEN
    CALL SendToGPIB (@params);
  ELSE
  IF request = ddSetStatus THEN
    DO;
      pSetStatus = params.pBuffer;
      IF setStatus.setStatMode = setTimeout
        THEN overflow.timeout = setStatus.setNewTime;
        ELSE error = notSupported;
    END;
  ELSE
  IF request = ddGetStatus THEN
    DO;
      CALL SETB (0, @getStatus, SIZE (getStatus));
      getStatus.open     = openStat;
      getStatus.access   = accessStat;
      getStatus.GPIBAddr = params.intAddr;
```

```
              IF params.length < SIZE (getStatus)
                THEN getStatusLength = params.length;
                ELSE getStatusLength = SIZE (getStatus);
              CALL MOVB (@getStatus, params.pBuffer, getStatusLength);
          END;
        ELSE
        IF request = ddRead THEN
        DO;
            /* If no address was assigned when attached,
               use the default address                    */
            IF params.intAddr = OFFH THEN
                params.intAddr = defaultAddress;
            params.pOverflow = @overflow;
            CALL OSCALLDriver  (@(5,'GPIB'),0,DOUBLE (ddRead),
                                @params,@error);
        END;
        ELSE
        IF NOT ((request = ddOpen)        OR
                (request = ddInitialize)  OR
                (request = ddClose)       OR
                (request = ddDetach)      OR
                (request = ddAttach)      OR
                (request = ddTruncate)    OR
                (request = ddDeactivate)) THEN
            error = notSupported;

END;

WriteString: PROC (pString) REENT;
  DCL pString PTR;
  DCL string  BASED pString STRUCTURE (len BYTE, chars (1) BYTE);
  DCL intParams ParamListType;

  intParams.pBuffer = @string.chars;
  intParams.length  = string.len;
  CALL SendToGPIB (@intParams);
END;

SendToGPIB: PROC (pParams) REENT;
  DCL pParams PTR;
  DCL params  BASED pParams ParamListType;
  DCL error   WORD;

  IF params.length > 0 THEN
    DO;
        /* If no address was assigned when attached,
           use the default address                    */
        IF params.intAddr = OFFH THEN
            params.intAddr = defaultAddress;
        params.pOverflow = @overflow;
        CALL OSCALLdriver (@(5,'GPIB'), 0, DOUBLE (ddWrite),
                           @params, @error);
    END;
END;
```

```
END; /* Module */
```

GENERIC GPIB SHELL WITH SERVICE REQUESTS EXAMPLE

```
$NOLIST LARGE OPTIMIZE(3)

/* Generic GPIB Shell with "hooks" for Service Requests
      Default GPIB address: 28.
      High Speed Mode.
*/

GenericGPIBDriver: DO;

$INCLUDE ('w0'Incs'PlmLit.Inc~Text~)
$INCLUDE (PrinterDriver.Inc~Text~)

/* Declarations for OS Procedures not in
    PrinterDriver.Inc~Text~  */

OsSignal: PROCEDURE (sid, mode, note, pError) EXTERNAL;
    DCL sid WORD;
    DCL mode BYTE;
    DCL note WORD;
    DCL pError PTR;
    END;

OsRegisterName: PROCEDURE (pName, token, mode, pError) EXTERNAL;
    DCL (pName, pError) PTR;
    DCL token DWORD;
    DCL mode BYTE;
    END;

OsWait: PROCEDURE (sid,time,pError) WORD EXTERNAL;
DCL sid WORD,
    time WORD,
    pError PTR;
END;

OsCreateSemaphore: PROCEDURE (pError) WORD EXTERNAL;
DCL pError PTR;
END;

OsDeleteSemaphore: PROCEDURE (sid,pError) EXTERNAL;
DCL sid WORD,
    pError PTR;
END;

DCL defTimeout     LIT '5000';   /* five seconds */
DCL defaultAddress LIT '28';
DCL HiSpeedMode    LIT '0';

DCL overflow OverflowType INITIAL (HiSpeedMode, OFFH, OFFH,
                                        defTimeout);

DCL firstimeThru BYTE INITIAL (OFFH);
```

```
DCL Note WORD;

DCL gateWaySetStatus STRUCTURE (mode      BYTE,
                                dataWord  WORD);

OsDevice: PROCEDURE (request, pParams, pError) PUB REENT;
  DCL request WORD;
  DCL pParams PTR;
  DCL pError  PTR;

  DCL error  BASED pError  WORD;
  DCL params BASED pParams ParamListType;

  DCL pSetStatus PTR;
  DCL setStatus BASED pSetStatus SetStatusType;
  DCL StatusBlock SetStatusType;
  DCL getStatus GetStatusType;
  DCL getStatusLength WORD;  ·

  IF request = ddWrite THEN
    CALL SendToGPIB (@params);
  ELSE
  IF request = ddSetStatus THEN
    DO;
      pSetStatus = params.pBuffer;
      IF setStatus.setStatMode = setTimeout
        THEN overflow.timeout = setStatus.setNewTime;
        ELSE error = notSupported;
    END;
  ELSE
  IF request = ddGetStatus THEN
    DO;
      CALL SETB (0, @getStatus, SIZE (getStatus));
      getStatus.open     = openStat;
      getStatus.access   = accessStat;
      getStatus.GPIBAddr = params.intAddr;
      IF params.length < SIZE (getStatus)
        THEN getStatusLength = params.length;
        ELSE getStatusLength = SIZE (getStatus);
      CALL MOVB (@getStatus, params.pBuffer, getStatusLength);
    END;
  ELSE
  IF request = ddRead THEN
  DO;
      IF error = 0 THEN         .
      DO;
        IF params.intAddr = OFFH THEN
           params.intAddr = defaultAddress;
        params.pOverflow = @overflow;

        CALL OSCALLDriver   (@(5,''GPIB'),0,DOUBLE (ddRead),
                             @params,@error);
      END;
  END;
```

```
        ELSE IF request = ddInitialize THEN
    DO;

/*  Tell the GPIB gateway driver to respond to SRQ for this device. */

        IF params.intAddr = OFFH THEN
            params.intAddr = defaultAddress;

        IF firstimeThru THEN
        DO;
            gateWaySetStatus.mode = 2; /* Set SRQ */

            /* Create the semaphore that the gateway driver
               will Signal when a SRQ is asserted by the
               device.        */

            gateWaySetStatus.dataWord =
                        OsCreateSemaphore (@error);

            IF error = 0 THEN
            DO;
              params.pOverflow = @gateWaySetStatus;
              CALL OSCALLDriver (@(5,''GPIB'),
                                 0,DOUBLE (ddSetStatus),
                                 @params,@error);
              firstimeThru = FALSE;

            /* Register the semaphore ID so an application can
               wait for a signal to this semaphore.  */

              CALL OsRegisterName (@(8,'gpibSema'),
                                   gatewaysetstatus.dataword,
                                   1,
                                   @error);

        END;
      END;
    END;
    ELSE IF request = ddDeactivate THEN
    DO;
        IF params.intAddr = OFFH THEN
            params.intAddr = defaultAddress;
        params.pOverflow = @overflow;

        /* Tell the gateway driver to stop responding to
           Service Requests from this device.            */
        CALL OSCALLDriver (@(5,''GPIB'),0,DOUBLE (ddDeactivate),
                        @params,@error);

        CALL OsDeleteSemaphore (gatewaysetstatus.dataWord,@error);

        /* Remove the name from the name table */
        CALL OsRegisterName (@(8,'gpibSema'),
                             gatewaysetstatus.dataword,
```

```
                                    2,
                                @error);

        END;
        ELSE
        IF NOT ((request = ddOpen)          OR
                (request = ddClose)         OR
                (request = ddDetach)        OR
                (request = ddAttach)        OR
                (request = ddTruncate)) THEN
          error = notSupported;

END;

WriteString: PROC (pString) REENT;
  DCL pString PTR;
  DCL string  BASED pString STRUCTURE (len BYTE, chars (1) BYTE);
  DCL intParams ParamListType;

  intParams.pBuffer = @string.chars;
  intParams.length  = string.len;
  CALL SendToGPIB (@intParams);
END;

SendToGPIB: PROC (pParams) REENT;
 .DCL pParams PTR;
  DCL params  BASED pParams ParamListType;
  DCL error    WORD;

  IF params.length > 0 THEN
    DO;

      IF error = 0 THEN
      DO;
        IF params.intAddr = OFFH THEN
          params.intAddr = defaultAddress;
        params.pOverflow = @overflow;
        CALL OSCALLdriver (@(5,''GPIB'), 0,
                            DOUBLE (ddWrite), @params,
                            @error);
    END;
    END;
END;

END; /* Module */
```

GENERIC GPIB DEVELOP FILE

```
:Name:      GenericGPIB
:Prefix:    GenericGPIB
:Listings:  'WO'LST'
:Objects:   'WO'OBJ'

:Sources:
  GenericGPIB.Plm
  GenericGPIBSRQ.Plm

:Control Yes w/Debug: DEBUG

:Link GenericGPIB:
LINK
'wO'OBJ'GenericGPIB.PLM~OBJ~,'wO'Libs'JmpDev.Asm~Obj~,'wO'Libs'LargeSingl
eException.Asm~Obj~,'wO'Libs'LargeSystemCalls~Lib~ TO GenericGPIB~Device~
BIND SS(STACK(O)) PC(PURGE) FASTLOAD PURGE
PRINT('wO'LST'GenericGPIB~MP1~)

:Link GenericGPIB SRQ:
LINK
'wO'OBJ'GenericGPIBSRQ.PLM~OBJ~,'wO'Libs'JmpDev.Asm~Obj~,'wO'Libs'LargeSi
ngleException.Asm~Obj~,'wO'Libs'LargeSystemCalls~Lib~ TO
GenericGPIBSRQ~Device~ BIND SS(STACK(O)) PC(PURGE) FASTLOAD PURGE
PRINT('wO'LST'GenericGPIBSRQ~MP1~)

:Test GenericGPIB:
  Deactivate GenericGPIB
  Activate GenericGPIB 28

:Test GenericGPIB SRQ:
  Deactivate GenericGPIBSRQ
  Activate GenericGPIBSRQ 28

:Command Line:
  'DevelopmentExecutive'

:GRiDManager:
  'GRiDManager'
```

APPENDIX A: UNIVERSAL PRINTER LANGUAGE

This appendix defines the GRiD Universal Printer Interface Language. It defines the interface between any application which supports printing and any GRiD supported printer driver.

Applications should attach to 'Printer. GRiD-OS will be responsible for mapping this into the current system printer. When opened, the driver initializes the printer into its normal typeface (12 pitch). When closed, the driver flushes the printer's internal buffer.

The commands which each printer must support (support can mean to ignore) are listed below. The driver must accept these codes in a serial byte stream.

```
Command              Turn on/off

Pass an ESC thru     ESC ESC
Boldface             ESC 'B'
Underline            ESC 'U'
Italics              ESC 'I'
Superscript          ESC '+'
Subscript            ESC '-'
Enlarged             ESC 'E'
Condensed            ESC 'C'

Line spacing         ESC 'L' n
  Where n is in 1/8ths of an line. ( 8/8 = single line spacing )

Graphics             ESC 'G'  width  height  topLeft.x  topLeft.y  nextRow
```

NOTE: When using sub or superscripts, the line spacing must be set to a value greater than single line spacing to allow room.

NOTE: Turning subscripting (superscripting) on automatically turns superscripting (subscripting) off.

In addition, each driver must support a command to print screen images. This is impossible for letter quality printers but they must skip the appropriate amount of "white space" so that an image could be pasted in later.

ESC 'G' width height topLeft.x topLeft.y nextRow

The five parameters are word values and are explained in the following diagram. All parameters are in pixels units. The values to print a normal screen image are:

    320, 240, 0, 0, 320

APPENDIX B: ERROR CODES

This section lists the error codes that can result from calling a gateway driver.

Serial Gateway Driver

0:   Everything OK. No action necessary.

35: Request Not Supported. You asked the gateway driver to do something impossible. For example, the serial driver does not support a seek request.

231: Device Not Active. This error can occur in three situations:
1. The serial device has not been activated. Be sure you have activated Serial~Device~ from the command line or programmatically.
2. You attempted to transfer data before establishing a connection. Make an OsSetStatus mode six call to establish a connection.
3. The gateway driver lost CTS and/or DCD. You can determine which one by making a OsGetStatus call and examining the modemControl byte. Make sure these signals are active on your hardware or make an OsSetStatus mode 60 call to change the required signals.

235: Bad Parameter. You passed the gateway driver a valid request but with a bad parameter. For example, you would get this error if you made an OsSetStatus mode 44 call to the serial gateway driver.

401: Time Out Error. This error can occur in two situations.
1. When reading, the gateway driver did not receive a character within the character time out period. Check your hardware or make an OsSetStatus mode two call to change the character time out.
2. When establishing a connection, the gateway driver did not detect a handshake within the connect time out period. Check

your hardware. You can make an OsSetStatus mode two call to change the connect time out period, or you can make an OsSetStatus mode 60 call to change the required signals.

402: Carrier Was Lost. Check your hardware.

403: Parity Error. Make an OsSetStatus mode one request to change parity type.

Modem Gateway Driver

0: Everything OK. No action necessary.

35: Request Not Supported. You asked the gateway driver to do something impossible. For example, the modem driver does not support a seek request.

231: Device Not Active. This error can occur in two situations:
1. The modem device has not been activated. Be sure you have activated Modem~Device~ from the command line or programmatically.
2. You attempted to transfer data before establishing a connection. Make an OsSetStatus mode six call to establish a connection.

235: Bad Parameter. You passed the gateway driver a valid request but with a bad parameter. For example, you would get this error if you made an OsSetStatus mode 61 call to the modem gateway driver.

400: Modem Did Not Answer. When establishing a connection, the gateway driver did not detect a handshake within the connect time out period. You can change the connect time out period with OsSetStatus mode two.

401: Time Out Error. When reading, the gateway driver did not receive a character within the character time out period. You can adjust the character time out period with OsSetStatus mode two.

402: Carrier Was Lost.

403: Parity Error. Make an OsSetStatus mode one request to change parity type.

406: Bad Phone Number. You asked the modem driver to dial a number that contained illegal characters. Only numbers or format characters are allowed. If you are using touchtone dialing, the "#" or "*" characters are also allowed.

GPIB Gateway Driver

0: Everything OK. No action necessary.

35: Request Not Supported. You asked the gateway driver to do something impossible. For example, the GPIB driver only supports set status modes of zero or two. Other requests will return this error.

451: GPIB Time Out. When reading, the gateway driver did not receive a character within the time out period. If you are using the low speed mode, you can adjust the time out period in the parameter block.

452: GPIB Not Responding. The gateway driver could not communicate with the other device. Be sure the other device is turned on and ready.